6502 Obfuscation

Intro

This document looks at techniques for obfucating 6502 machine code.

Aims of obfuscation

- Hinder disassembly of the machine code (i.e. conversion into assembley code)
- Make disassembled code harder to follow and reason about

Why do people obfuscate?

Various possible reasons:

- Stop other people using your techniques or data structures
- Stop people claiming a prize falsely (e.g. a game that gives a competition code on completion)
- Stop people making cheats for games

Obfuscation: the facts

Obfuscation is only obfuscation. You're only making it harder, but not impossible, for someone with the target hardware or emulator to see the target code.

Levels of Obfuscation

1. None
2. Feasible to unpick by hand (simple things like self-altering code that changes a few memory locations)
3. Requires running on architecture emulator (e.g. a 6502 simulator) as too unwieldy for handiwork
4. Requires running on actual hardware or accurate emulator, as depends on platform specific things, e.g. data in a BBC Micro system that is external to the code being attacked

Obfuscation Techniques

Non-desctructive padding

Pad legitimate code with non-destructive operations that make it look like a data section. e.g. several CLCs in a row, NOPs, etc. This code still executes a useful function but wastes memory/cycles on doing nothing of use, in the hope a viewer will dismiss it as “not real code”.

Code that is never reached

Code can be included that is ‘fake’, i.e. it is never executed (and never needs to be). In order to draw attention to the fake code, you can execute some code self-modification on the fake code in order to make it look like it’s relevant and will later be called.

Disassembler mis-alignment

You can wrong-foot mis-align the disassembler by interrupting flow of opcodes with e.g. ascii or nonsense byte(s). These memory locations are never actually reached during execution so won’t cause an execution problem.

For example, after an RTS you might place a single byte $69. The location after the $69 is the true start of the next routine. The disassembler recognises $69 as ADC immediate mode, and expects a byte to follow as the parameter. Thus the disassembler is mis-aligned from this point, and outputs nonsense.

Obviously, you should never attempt to jump to the byte containing the $69!

Code self-modification

Modifying byte by byte

Part of the machine code can alter other parts of the code in order to disguise opcodes, their parameters, or data.

This can be done on a byte-by-byte basis. For example, disguising JSR or JMP parameters is a good way to obscure the flow of the machine code. Or you might change the opcode itself.

As a rule, if your code self-modifies to fix a deliberately bogus opcode to the correct value, the before and after opcode values should be for instructions with different byte lengths, as this will introduce misalignment chaos into a disassembly attempt.

For example, if your code overwrites a bogus opcode value with the opcode for LDA Absolute,Y, which is an instruction 3 bytes long, the bogus opcode should represent an instruction with length other than 3. This can introduce more disassembler mis-alignment.

Scrambling code sections

Alternatively, entire sections of machine code can be scrambled. Once the machine code is launched, it unscrambles these sections.

Modifying code sections just before use

To go a step further, certain sections of code can be descrambled just before they are excuted, then scrambled again. This would prevent someone doing a memory dump of an emulator once the code is running (which they'd do in an attempt to see the machine code once it's all unscrambled).

Modifying code sections - recursively

Image an unscrambling routine, '1', that unscrambles a section of memory. Then a different unscrambling routine '2', which is at the start of that unscrambled block, can then do the same trick again ad nauseam.

For maximum effect, employ this trick on a piece of code that is important but also only needed periodically, and use the 'Modifying code sections just before use' technique.

Use 6502 optimisation techniques to distract

There are 6502 optimisation techniques that can make code looks slightly odd, e.g. using JMPs in place JSRs to avoid tail RTS. This oddness can distract from the real flow of the code, or make the viewer think they're seeing pretend code.

Scrambling parameters: external source

Iinstead of hardcoding scrambling parameters into the machine code, derive them from a (definitely known!) external memory location in the target system, e.g. BBC micro. If the contents depend on the current context (e.g. we're in MODE2), even better.

You have to be sure that the immutability of the data you're using is certain!

Other possible tricks

Opcode alignment

Have all JSRs which start on (say) odd numbered memory locations be bogus, and then a startup routine can go and write NOP over them (or some other code that performs some harmless no-effect action).

In order to align JSRs where you want them, you can use a NOP or other pointless single byte instruction somewhere before, if necessary.

Make the code break if any part modified

So, for example, offset JMPs and JSRs using a routine that has summed all the bytes of the code and stored it somewhere. Any attempt to modify the code will break things.

Examples strategy combining the above tricks

Todo